IF184401_DAA(H)_FIN_5025211196_Sandyatama Fransisna Nugraha

Final Exam Report

1. We will be started with easy questions. It's about True (T) or false (F) questions. [15 points]
   - Quick-sort has worse asymptotic complexity than merge-sort (T/F). **(F)**
   - Binary search is O (log n) (T/F). **(T)**
   - Linear search in an unsorted array is O (n) , but linear search in a sorted array is O (log n). **(F)**
   - Linear search in a sorted linked list is O (n). **(T)**
   - If you are only going to look up one value in an array, asymptotic complexity favours doing a linear search on the unsorted array over sorting the array and then doing a binary search (T/F). **(T)**
   - If all arc weights are unique, the minimum spanning tree of a graph is unique (T/F). **(T)**
   - Binary search in an array requires that the array is sorted (T/F). **(T)**
   - Insertion into an ordered list can be done in O (log n) time (T/F) **(F)**
   - A good hash function tends to distribute elements uniformly throughout the hash table (T/F). **(T)**
   - In practice, with a good hash function and non-pathological data, objects can be found in O (1) time if the hash table is large enough (T/F). **(T)**
   - If a piece of code has asymptotic complexity O (g(n)), then at least g(n) operations will be executed whenever the code is run with parameter n (T/F). **(T)**
   - Given good implementations for different algorithms for some processes such as sorting, searching or finding a minimum spanning tree, you should always choose the algorithm with the better asymptotic complexity (T/F). **(T)**
   - It is not possible for the depth-first and breadth-first traversal of a graph to visit nodes in the same sequence if the graph contains more than two nodes (T/F). **(F)**
   - The maximum number of nodes in a binary tree of height H (H = 0 for leaf nodes) is 2H+1 – 1 (T/F). **(T)**
   - In a complete binary tree, only leaf nodes have no children (T/F). **(F)**

2. Please write down a polynomial-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. Please explain in detail your assumption, the pseudo-code of this algorithm, and explain the time complexity of this algorithm. (Assume that each integer appears once in the input sequence of n numbers) [20 points]

Answer

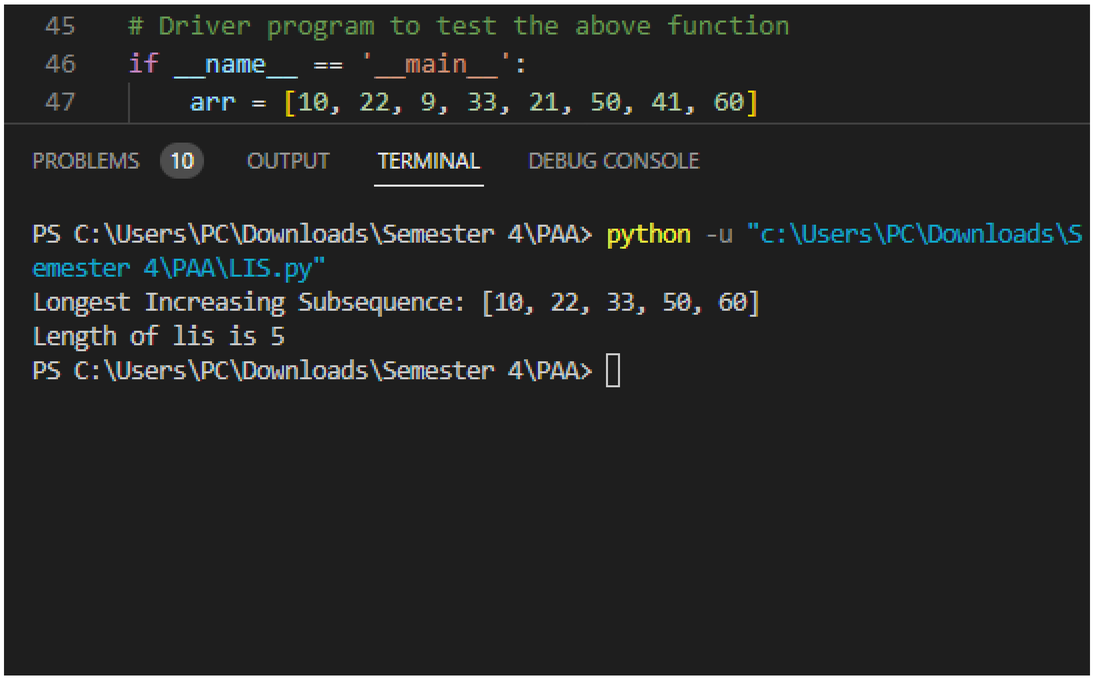
- Detail of the algorithm

The problem of finding the longest increasing subsequence of a sequence of n numbers involves finding the longest subsequence of numbers that are strictly increasing from left to right. This problem can be solved using dynamic programming or the patience sorting algorithm. The LIS algorithm is a commonly used approach that employs binary search to find the position of the smallest element that can end an increasing subsequence of a given length.

- Full-code (https://www.geeksforgeeks.org/longest-increasing-subsequence-dp-3/)

```
# Global variable to store the LIS sequence
global lis_sequence

def _lis(arr, n):
    # To allow access to the global variable
    global lis_sequence

    # Base Case
    if n == 1:
        return [arr[0]]

    # lisEndingHere is the LIS ending with arr[n-1]
    lisEndingHere = [arr[n-1]]

    # Recursively get all LIS ending with arr[0], arr[1]..arr[n-2]
    for i in range(1, n):
        res = _lis(arr, i)
        if arr[i-1] < arr[n-1] and len(res) + 1 > len(lisEndingHere):
            lisEndingHere = res + [arr[n-1]]

    # Compare lisEndingHere with the overall LIS sequence
    if len(lisEndingHere) > len(lis_sequence):
        lis_sequence = lisEndingHere

    return lisEndingHere

def lis(arr):
    # To allow access to the global variable
    global lis_sequence

    # Length of arr
    n = len(arr)

    # Initialize the LIS sequence with the first element
    lis_sequence = [arr[0]]

    # The function _lis() stores its result in lis_sequence
    _lis(arr, n)

    return lis_sequence

# Driver program to test the above function
if __name__ == '__main__':
    arr = [10, 22, 9, 33, 21, 50, 41, 60]
    n = len(arr)

    # Function call
    lis_seq = lis(arr)
    print("Longest Increasing Subsequence:", lis_seq)

```

Result :



```
# Driver program to test the above function
if __name__ == '__main__':
    arr = [10, 22, 9, 33, 21, 50, 41, 60]
```

```
PS C:\Users\PC\Downloads\Semester 4\PAA> python -u "c:\Users\PC\Downloads\S
emester 4\PAA\LIS.py"
Longest Increasing Subsequence: [10, 22, 33, 50, 60]
Length of lis is 5
PS C:\Users\PC\Downloads\Semester 4\PAA> 
```

- Pseudocode (https://www.baeldung.com/cs/longest-increasing-subsequence-dynamic-programming)

```
Algorithm 2: DP implementation
  Data: Input array of size n
  Result: LIS
  Lis[n]:= 1,1,1,1,....,1;
  for $i$=2...$n$ do
      for $j$=1...$i$-1 do
          if $array[i] > array[j]$ AND $Lis[i] \leq Lis[j]$ then
              Lis[i]:= Lis[j] + 1;
          end
      end
  end
  Return MAX(Lis[0], Lis[1], ..., Lis[n]);
```

Here's the step-by-step procedure:

1. Initialize lis[i] = 1 for every index i of the input array arr. This is because every element is a subsequence of length 1 on its own.
2. Iterate over the input array arr, from left to right, and for every element arr[i], iterate over all previous elements arr[j] where j < i.
3. For each j < i, compare arr[j] to arr[i]. If arr[i] is greater than arr[j], then we have a potential LIS ending at index i that includes arr[j]. We check if including arr[j] in this LIS would result in a longer LIS than the current value of lis[i] (which is currently set to 1). If lis[j] + 1 > lis[i], we update lis[i] to lis[j] + 1, since we have found a longer LIS ending at index i that includes arr[j].
4. After iterating over all previous elements arr[j] for each i, we have computed the length of the LIS ending at each index of the array. We return the maximum value of lis as the length of the longest increasing subsequence.
5. To reconstruct the actual LIS, we can iterate over the array again and store the indices where lis[i] is equal to the maximum LIS value. We then use these indices to build the LIS by iterating over the input array arr and adding the corresponding elements to the LIS.

- Time complexity

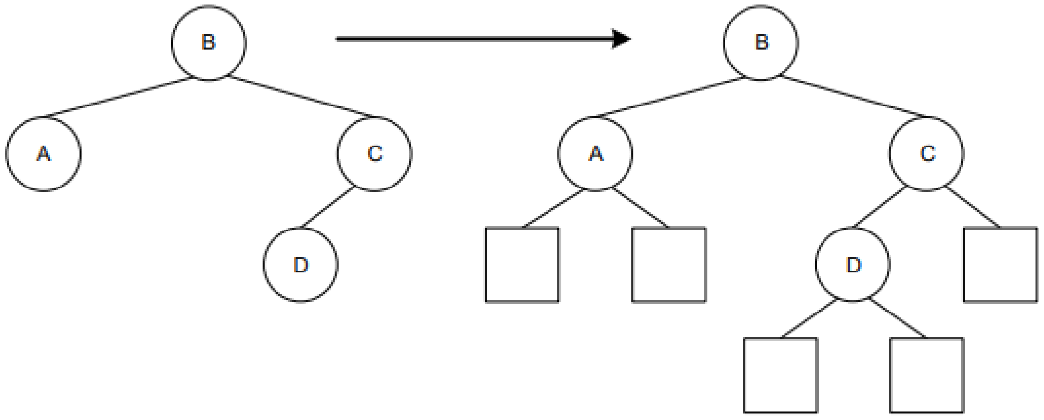
The time complexity of the dynamic programming algorithm for finding the longest monotonically increasing subsequence of a sequence of n numbers is O(n^2). To see why this is the case, let's break down the steps of the algorithm and analyze the time complexity of each step:

1. Initializing the array of longest increasing subsequences to 1 for each index takes O(n) time.
2. The most time-consuming step is the nested loop that compares each pair of indices in the input sequence. For each index i, we compare it with all the preceding indices j < i. This requires O(n^2) time in the worst case.
3. After filling out the array of longest increasing subsequences, we need to find the maximum value in the array, which takes O(n) time.
4. We then need to identify all the indices where the value in the array is equal to the maximum value. This requires iterating over the array again, which takes O(n) time.
5. Finally, we extract the longest increasing subsequence by iterating over the identified indices and extracting the corresponding numbers from the input sequence. This takes at most O(n) time.

Overall, the time complexity of the algorithm is dominated by the nested loop, which takes O(n^2) time. Therefore, the total time complexity of the algorithm is O(n^2). It's worth nothing that there are more efficient algorithms for finding the longest increasing subsequence, such as the Patience Sorting algorithm, which has a time complexity of O(n log n). However, the dynamic programming algorithm is conceptually simpler and easier to implement, and it can handle input sequences of practical size quite efficiently.

3. After all, binary trees have been discussed a lot in our lecture. It’s often used in search applications, where the tree is searched by starting at the root and then proceeding either to the left or the right child, depending on some condition. In some instances, the search stops at a node in the tree. However, in some cases, it attempts to cross a null link to a non-existent child. The search is said to “fall out” of the tree. The problem with falling out of a tree is that you have no information about where this happened, and often this knowledge is useful information.

   Given any binary tree, an extended binary tree is formed by replacing each missing child (a null pointer) with a special leaf node, namely an external node. The remaining nodes are called internal nodes. An example is shown in the figure below, where the external nodes are shown as squares and the internal nodes are shown as circles. Note that if the original tree is empty, the extended tree consists of a single external node. Also, observe that each internal node has exactly two children and each external node has no children. Let n denote the number of internal nodes in an extended binary tree. An extended binary tree with n internal nodes has n + 1 external node. Prove this statement by induction. [20 points]



Answer :

- Detail of the solution

The statement to be proved is: "An extended binary tree with n internal nodes has n + 1 external nodes." The proof is to be done by induction. Before we begin, we need to have an understanding of what an extended binary tree is. An extended binary tree is formed by replacing each missing child (a null pointer) with a special leaf node, namely an external node. The remaining nodes are called internal nodes. If the original tree is empty, the extended tree consists of a single external node. Also, observe that each internal node has exactly two children and each external node has no children.

- Strong Induction

To prove that an extended binary tree with n internal nodes has n + 1 external nodes, we will use strong induction. An extended binary tree is formed by replacing each missing child with an external node, while the remaining nodes are internal nodes. Each internal node has exactly two children, and each external node has no children. If the original tree is empty, the extended tree consists of a single external node. The assumption is especially useful for showing the recursive algorithm work.

1. We state the proposition P(n) that we want to prove: an extended binary tree with n internal nodes has n + 1 external nodes.

2. We prove the base case P(0), which is that an extended binary tree with 0 internal nodes has 1 external node. This is straightforward, since an extended binary tree with 0 internal nodes is simply an external node.
3. Now move on to the inductive step. We assume that P(n) holds for all values of n up to some integer k. That is, we assume that an extended binary tree with k internal nodes has k+1 external nodes.
4. We need to prove that P(k+1) is true, i.e., that an extended binary tree with k+1 internal nodes has k+2 external nodes. To do this, we can start with an extended binary tree with k internal nodes and add a new internal node as the parent of two external nodes. This increases the number of internal nodes by 1 and the number of external nodes by 2. Therefore, the total number of external nodes in the new tree is (k+1)+2 = k+3.
5. However, we need to subtract 1 from this value to account for the fact that one of the external nodes used to create the new internal node is no longer an external node. Therefore, the total number of external nodes in the new tree is (k+1)+2-1 = k+2, which is exactly what we wanted to show.

Finally, we conclude by strong induction that P(n) holds for all n ≥ 0. Therefore, an extended binary tree with n internal nodes has n+1 external nodes for all non-negative integers n.

```
/** Return an index of x in a.
* Requires: a is sorted in ascending order, and x is found in the
array a somewhere between indices left and right. */

int binsrch(int x, int[] a, int left, int right) {
        int m = (left+right)/2;
        if (x == a[m]) return m;
        if (x < a[m]) return find(x, a, l, m−1)
        else
        return find(x, a, m+1, r);
}

/* Because this code is tail-recursive, we can also transform it into iterative code straightforwardly */

int binsrch(int x, int[] a, int left, int right) {
while (true) {
        int m = (left+right)/2;
        if (x == a[m]) return m;
        if (x < a[m]) r = m−1;
        else l = m+1;
}
```

Binary search is a popular algorithm that is easy to understand and implement, but it is also notoriously prone to errors. Merely testing the code may not be sufficient to uncover all the cases in which it fails. To ensure that binary search works correctly, we can prove its correctness using induction. The recursive version of the algorithm is a good candidate for this proof technique. However, because binary search calls itself, the argument can become circular, which we need to avoid. The key observation is that binary search works in a divide-and-conquer fashion, where the arguments become smaller in some way. The variable that represents the decrease in arguments is the difference between the right and left parameters. We can use this variable to construct our inductive argument.

To prove the correctness of binary search, we can follow the strong induction recipe. We start by stating the proposition P(n) that binary search works correctly for inputs where right-left = n. We then prove the base case P(0), which states that binary search works when the range is a single element. For the inductive step, we assume that binary search works when the range is smaller than or equal to k, and we aim to prove that it works when the range is k+1. There are three cases to consider, depending on whether the target element is less than, equal to, or greater than the middle element of the range. In each case, we show that the recursive call is made with a smaller range of values, and we use the induction hypothesis to conclude that the recursive call works correctly. Because the inductive step works in all cases, we can conclude that binary search is correct.

Similarly, we can prove the claim that a binary tree with n nodes has n+1 null child links using induction. We define x(n) to be the number of null child links in a binary tree of n nodes. We show that x(n) equals n+1 for all n>0 by proving the base case x(0)=1 and the inductive step x(n)=x(nL)+x(nR)=(nL+1)+(nR+1)=n+1, where nL and nR are the numbers of nodes in the left and right subtrees, respectively. This proof shows the power of inductive reasoning in establishing the correctness of algorithms and data structures.

4. We assumed that G = (V, E) be a complete graph with the set of vertices V = {1, 2, …, n} and weights w(i, j) = dij where dij isthe distance from the input of Traveling Salesman Problem (TSP). Then we can do the following steps. (1) Select a vertex $r \in V$ to serve as a root for the future tree. (2) Build a minimum spanning tree T for G with the root r using a polynomial-time minimum spanning tree algorithm as a subroutine. (3) Construct the list L of vertices being a Pre-order Walk of T. (4) Return L as an approximate tour of G.

   Please prove the following statements.

   - The approximation algorithm for Travelling Salesman with triangle inequality is correct (i.e., produces a tour). [5 points]

- The running time of the algorithm is polynomial. [5 points]
- The algorithm has a ratio bound ρ(n) = 2. [10 points]

Answer :

- The approximation algorithm for Travelling Salesman with triangle inequality is correct (i.e., produces a tour). [5 points]

To prove the correctness of the approximation algorithm for the Traveling Salesman Problem (TSP) with triangle inequality, we need to show that it produces a valid tour of the input graph. The algorithm starts by selecting a root vertex r and building a minimum spanning tree T of the complete graph G with r as the root using a polynomial-time minimum spanning tree algorithm. Since T is a tree, it is a connected graph that spans all the vertices of G and does not contain any cycles. Therefore, T is a valid tour of the input graph.

However, a tour must be a connected graph that contains a cycle that visits each vertex exactly once. To obtain a tour from the minimum spanning tree T, we can perform a depth-first search (DFS) on T starting from the root vertex r and record the order in which the vertices are visited. This order gives us a Hamiltonian path in T. To obtain a Hamiltonian cycle, we simply add an edge from the last vertex visited back to the root vertex r. This gives us a tour of the complete graph G that visits each vertex exactly once.

The approximation algorithm then constructs a pre-order walk of T and returns it as an approximate tour of G. Since the pre-order walk of a tree simply rearranges the order in which the vertices are visited, it does not affect the validity of the tour. Therefore, the tour obtained from the pre-order walk of T is also a valid tour of the input graph.

In summary, the approximation algorithm for TSP with triangle inequality is correct because it produces a valid tour of the input graph. The algorithm constructs a minimum spanning tree T of the input graph and then obtains a Hamiltonian cycle from T by performing a DFS and adding an edge back to the root vertex. The algorithm then constructs a pre-order walk of T, which gives us a rearranged version of the minimum spanning tree T as an approximate tour of the input graph. Since T spans all the vertices of G and the pre-order walk of T simply rearranges the order in which the vertices are visited, the tour obtained from the pre-order walk of T is also a valid tour of the input graph.

- The running time of the algorithm is polynomial. [5 points]

To prove that the running time of the algorithm for the Traveling Salesman Problem (TSP) with triangle inequality is polynomial, we need to analyze the time complexity of each step of the algorithm. The input graph is a complete graph G = (V, E) with the set of vertices V = {1, 2, …, n} and weights w(i, j) = dij, where dij is the distance from the input of TSP.

The first step is to select a vertex r to serve as the root of the future tree, which takes constant time and does not affect the overall time complexity of the algorithm.

The second step is to build a minimum spanning tree T for G with the root r using a polynomial-time minimum spanning tree algorithm as a subroutine. The time complexity of this step depends on the specific minimum spanning tree algorithm used. However, all commonly used algorithms, such as Prim's algorithm, Kruskal's algorithm, and Boruvka's algorithm, have a time complexity of O(m log n) or O(m log m), where m is the number of edges in the graph and n is the number of vertices. Since G is a complete graph, it has m = n(n-1)/2 edges, and therefore the time complexity of this step is O(n^2 log n) or O(n^2 log (n^2)), which is polynomial in n.

The third step is to construct the list L of vertices by performing a Pre-order Walk of T, which takes linear time as each vertex is visited exactly once. Therefore, the time complexity of this step is O(n).

The fourth and final step is to return L as an approximate tour of G, which takes constant time and does not affect the overall time complexity of the algorithm.

Thus, the total time complexity of the algorithm is the sum of the time complexities of the individual steps, which is O(n^2 log n) or O(n^2 log (n^2)), depending on the minimum spanning tree algorithm used. However, if we use a polynomial-time minimum spanning tree algorithm such as Kruskal's algorithm or Prim's algorithm, the running time of the overall algorithm will be polynomial. For instance, Kruskal's algorithm has a running time of O(m log n), where m is the number of edges in the graph and n is the number of vertices. In the case of TSP, the input graph is a complete graph, which means that it has n(n-1)/2 edges. Therefore, the running time of Kruskal's algorithm for TSP will be O(n^2 log n), which is polynomial in n.

In conclusion, the algorithm for TSP with triangle inequality has a polynomial running time, as long as a polynomial-time minimum spanning tree algorithm is used as a subroutine.

- The algorithm has a ratio bound $\rho(n) = 2$. [10 points]

To prove that the algorithm for TSP with triangle inequality has a ratio bound of 2, we need to show that the length of the approximate tour returned by the algorithm is at most twice the length of the optimal tour. Let OPT be the length of the optimal tour, and let L be the length of the approximate tour returned by the algorithm.

First, note that the length of the minimum spanning tree T is a lower bound on the length of any tour that visits all vertices of G exactly once. This is because any tour that visits all vertices exactly once must include all edges of T, since T is a tree that spans all vertices of G. Therefore, we have OPT ≥ cost(T), where cost(T) is the length of T.

Next, note that the length of the approximate tour L is twice the sum of the edge weights of T, since each edge of T is visited twice during the Pre-order Walk. Therefore, we have:

L = 2 * cost(T)

Combining this with the inequality OPT ≥ cost(T), we get:

L ≤ 2 * OPT

Therefore, the length of the approximate tour returned by the algorithm is at most twice the length of the optimal tour, and the algorithm has a ratio bound of 2.

This ratio bound of 2 means that the approximate tour returned by the algorithm is guaranteed to be at most twice as long as the optimal tour. In practice, this means that the algorithm provides a good approximation to the optimal tour, especially for large input sizes where finding the exact optimal tour is computationally infeasible. Additionally, the algorithm has a polynomial running time, which makes it practical for solving real-world TSP instances.

In summary, the algorithm for TSP with triangle inequality has a ratio bound of 2, which guarantees that the approximate tour returned by the algorithm is at most twice as long as the optimal tour. This makes the algorithm a practical and effective tool for solving TSP instances, especially for large input sizes.

5. Bellman-Ford and Dijkstra are two algorithms to find the shortest path from one vertex to all other vertices of a weighted graph.

   Please do the following:

   - Describe their differences and their respective real-life applications. [10 points]
   - Write pseudocode for each algorithm and try to show where they differ. [10 points]
   - Create a short illustration of how each algorithm works. [5 points]

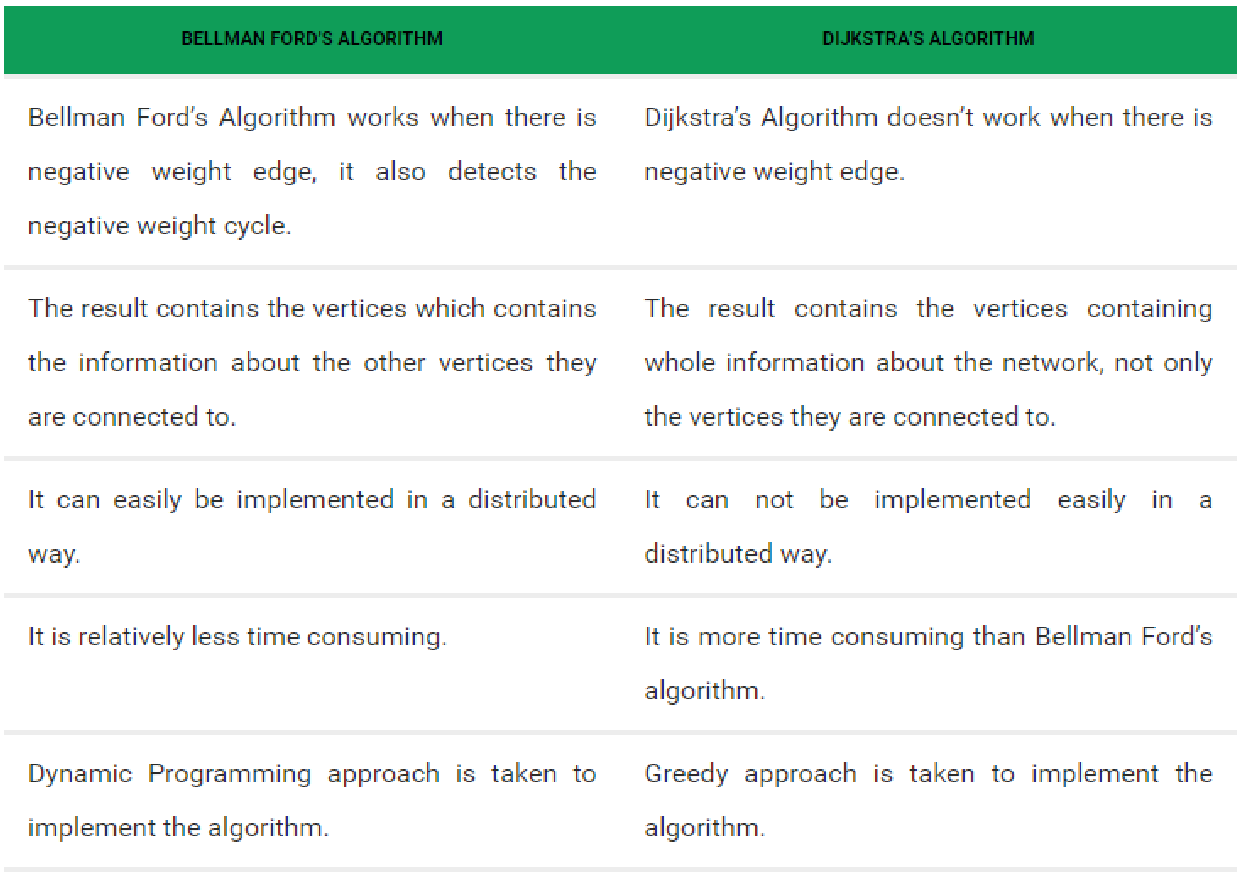
Answer :

1. The different between Bellman-Ford and Dijkstra algorithm

- Definition

The Bellman-Ford algorithm is used to find the shortest path between two vertices in a weighted graph. It works by repeatedly relaxing all the edges in the graph and updating the distance to the destination vertex if a shorter path is found. The algorithm can detect negative weight cycles, and its time complexity is O(|V||E|). Despite its slower time complexity, it is useful in dealing with graphs with negative weights, and it is widely used in network routing and computer networking.

The Dijkstra algorithm is used to find the shortest path between two vertices in a weighted graph. It works by selecting the vertex with the shortest distance from the source vertex and relaxing its neighboring vertices. The algorithm terminates when the destination vertex is visited or all unvisited vertices have infinite distance. Its time complexity is O(|E| + |V|log|V|) using a priority queue, and it is widely used in network routing, computer networking, and map routing applications. Unlike the Bellman-Ford algorithm, Dijkstra's algorithm cannot handle negative weight edges. To understanding completely, I am included the different in the table.



BELLMAN FORD'S ALGORITHM	DIJKSTRA'S ALGORITHM
Bellman Ford's Algorithm works when there is negative weight edge, it also detects the negative weight cycle.	Dijkstra's Algorithm doesn't work when there is negative weight edge.
The result contains the vertices which contains the information about the other vertices they are connected to.	The result contains the vertices containing whole information about the network, not only the vertices they are connected to.
It can easily be implemented in a distributed way.	It can not be implemented easily in a distributed way.
It is relatively less time consuming.	It is more time consuming than Bellman Ford's algorithm.
Dynamic Programming approach is taken to implement the algorithm.	Greedy approach is taken to implement the algorithm.

- Real Life Application

Both the Bellman-Ford algorithm and the Dijkstra algorithm are used in real-life applications that involve finding shortest paths in graphs. However, they are often used in different types of applications due to their strengths and weaknesses.

The Bellman-Ford algorithm is useful in applications that involve graphs with negative weight edges, as it can detect and handle negative weight cycles. This makes it a good choice for network routing problems involving traffic congestion or variable link costs. For example, it can be used to find the shortest path between two cities on a map with varying traffic conditions.

On the other hand, the Dijkstra algorithm is more efficient than Bellman-Ford in finding the shortest path in graphs with non-negative edge weights. This makes it a good choice for applications such as GPS navigation and map routing, where edge weights are fixed and non-negative. It is also commonly used in computer networking to calculate the distances between routers in a network.

- Conclusion

In summary, the choice between Bellman-Ford and Dijkstra depends on the specific problem requirements and the characteristics of the graph being analyzed. Bellman-Ford is more suitable for graphs with negative weights, while Dijkstra is more efficient for graphs with non-negative weights.

2. Write the preudocode

Implementation of Bellman-Ford Algorithm (https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/)

```
class Graph:

    # Initialisasion the graph
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    # function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # function to print the solution
    def printArr(self, dist):
        print("Vertex Distance from Source")
        for i in range(self.V):
            print("{0}\t\t{1}".format(i, dist[i]))

    # The main function that finds shortest distances from src to all other vertices using Bellman-Ford
algorithm
    def BellmanFord(self, src):
        dist = [float("Inf")] * self.V
        dist[src] = 0
        for _ in range(self.V - 1):
            for u, v, w in self.graph:
                if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                    dist[v] = dist[u] + w
        for u, v, w in self.graph:
            if dist[u] != float("Inf") and dist[u] + w < dist[v]:
                print("Graph contains negative weight cycle")
                return

        self.printArr(dist)

# Main Program
if __name__ == '__main__':
```

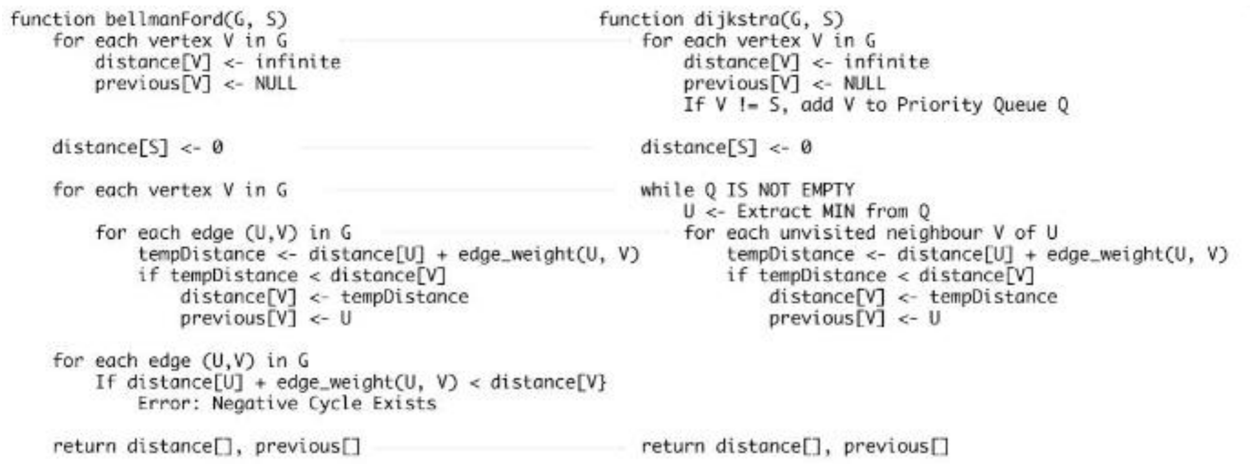
```
# Graph Declaration
g = Graph(5)
g.addEdge(0, 1, -1)
g.addEdge(0, 2, 4)
g.addEdge(1, 2, 3)
g.addEdge(1, 3, 2)
g.addEdge(1, 4, 2)
g.addEdge(3, 2, 5)
g.addEdge(3, 1, 1)
g.addEdge(4, 3, -3)

# Function call
g.BellmanFord(0)
```

Implementation of Dijkstra Algorithm (https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/)

```
# Library for INT_MAX
import sys


class Graph():

     # Initialisasion the graph
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    # Function to print the solution
    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])

    # Function to find the vertex with minimum distance value, from the set of vertices not yet included in
shortest path tree
    def minDistance(self, dist, sptSet):
        min = sys.maxsize
        for u in range(self.V):
            if dist[u] < min and sptSet[u] == False:
```

```
        min = dist[u]
        min_index = u

    return min_index

  # Function that implements Dijkstra's single source shortest path algorithm for a graph represented using
adjacency matrix representation
  def dijkstra(self, src):

    dist = [sys.maxsize] * self.V
    dist[src] = 0
    sptSet = [False] * self.V
    for cout in range(self.V):
      x = self.minDistance(dist, sptSet)
      sptSet[x] = True
      for y in range(self.V):
        if self.graph[x][y] > 0 and sptSet[y] == False and \
            dist[y] > dist[x] + self.graph[x][y]:
          dist[y] = dist[x] + self.graph[x][y]

    self.printSolution(dist)


# Main Program
if __name__ == "__main__":

  # Graph Declaration
  g = Graph(9)
  g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
      [4, 0, 8, 0, 0, 0, 0, 11, 0],
      [0, 8, 0, 7, 0, 4, 0, 0, 2],
      [0, 0, 7, 0, 9, 14, 0, 0, 0],
      [0, 0, 0, 9, 0, 10, 0, 0, 0],
      [0, 0, 4, 14, 10, 0, 2, 0, 0],
      [0, 0, 0, 0, 0, 2, 0, 1, 6],
      [8, 11, 0, 0, 0, 0, 1, 0, 7],
      [0, 0, 2, 0, 0, 0, 6, 7, 0]
      ]

  # Function call
  g.dijkstra(0)
```

Another pseudocode of the comparison between Bellman-Ford and Dijkstra Algorithm (https://www.programiz.com/dsa/bellman-ford-algorithm)



```
function bellmanFord(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL

    distance[S] <- 0

    for each vertex V in G

        for each edge (U,V) in G
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    for each edge (U,V) in G
        If distance[U] + edge_weight(U, V) < distance[V}
            Error: Negative Cycle Exists

    return distance[], previous[]
```

```
function dijkstra(G, S)
    for each vertex V in G
        distance[V] <- infinite
        previous[V] <- NULL
        If V != S, add V to Priority Queue Q

    distance[S] <- 0

    while Q IS NOT EMPTY
        U <- Extract MIN from Q
        for each unvisited neighbour V of U
            tempDistance <- distance[U] + edge_weight(U, V)
            if tempDistance < distance[V]
                distance[V] <- tempDistance
                previous[V] <- U

    return distance[], previous[]
```

Bellman-Ford algorithm works by initially overestimating the distance of each vertex from the source vertex, then iteratively relaxing these estimates until the shortest path is found. This algorithm is guaranteed to find the shortest path in a graph with no negative cycles. However, if the graph contains a negative cycle, the algorithm will detect it and report that no shortest path exists.

1. Bellman Ford algorithm is used to find the shortest path from a single source vertex to all other vertices in a weighted graph.
2. It works by iteratively relaxing the edges of the graph and updating the distance of each vertex from the source vertex until the shortest path is found.
3. Bellman Ford algorithm can handle negative edge weights and can detect negative weight cycles in the graph.
4. The time complexity of Bellman Ford algorithm is O(V * E), where V is the number of vertices and E is the number of edges in the graph.
5. Bellman Ford algorithm is slower than Dijkstra's algorithm for graphs with non-negative edge weights.

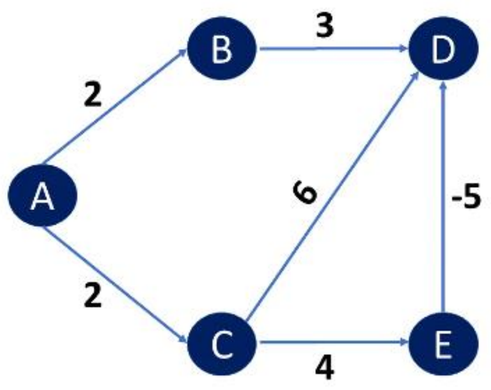
On the other hand, Dijkstra's algorithm is a greedy algorithm that works by finding the minimum distance vertex from the source vertex and relaxing its neighboring vertices. This process is iterated until all vertices are visited. Dijkstra's algorithm has a better time complexity than Bellman-Ford algorithm, especially for dense graphs with non-negative edge weights. However, it does not work for graphs with negative edge weights.

1. Dijkstra's algorithm is used to find the shortest path from a single source vertex to all other vertices in a weighted graph.
2. It works by maintaining a priority queue of vertices and their distances from the source vertex. It then selects the vertex with the minimum distance and updates the distances of its neighbors.
3. Dijkstra's algorithm can only handle non-negative edge weights and cannot detect negative weight cycles in the graph.
4. The time complexity of Dijkstra's algorithm is O((V+E)logV), where V is the number of vertices and E is the number of edges in the graph.
5. Dijkstra's algorithm is faster than Bellman Ford algorithm for graphs with non-negative edge weights.

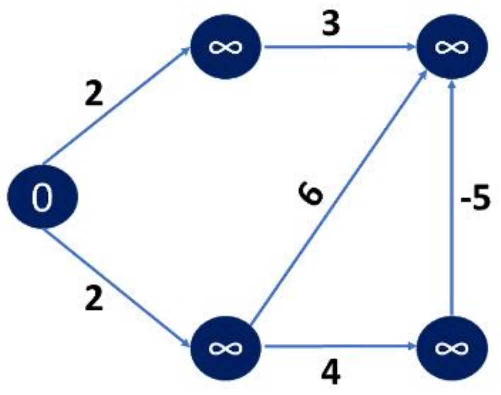
3. Create the ilustration of the code

Bellman-Ford Algorithm (https://www.simplilearn.com/tutorials/data-structure-tutorial/bellman-ford-algorithm)

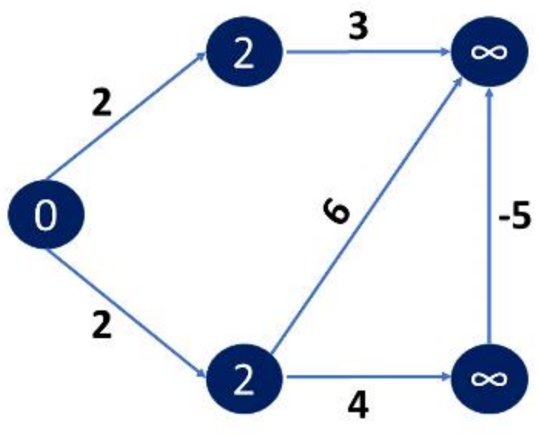
1. Consider the weighted graph below



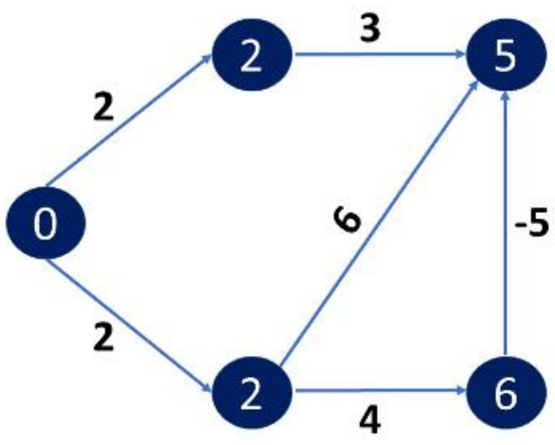
2. Choose path value 0 for the source vertex and infinity for all other vertices.



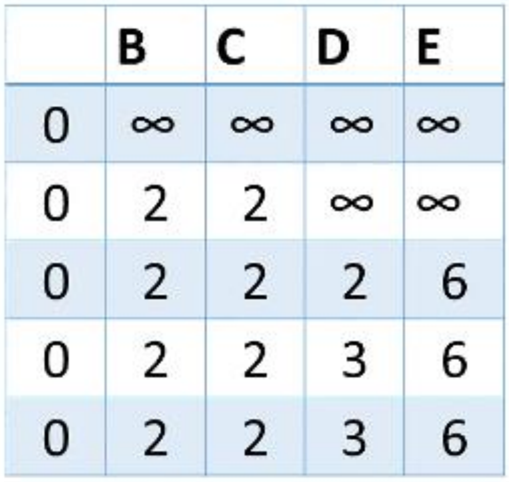
3. If the new calculated path length is less than the previous path length, go to the source vertex's neighboring Edge and relax the path length of the adjacent Vertex.



4. This procedure must be repeated V-1 times, where V is the number of vertices in total. This happened because, in the worst-case scenario, any vertex's path length can be changed N times to an even shorter path length.



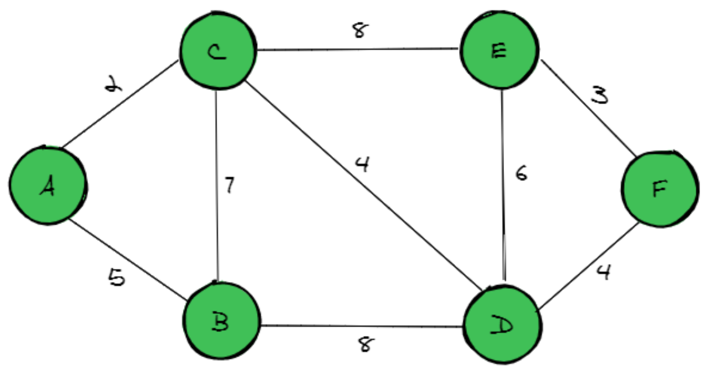
5. As a result, after V-1 iterations, you find your new path lengths and can determine in case the graph has a negative cycle or not.



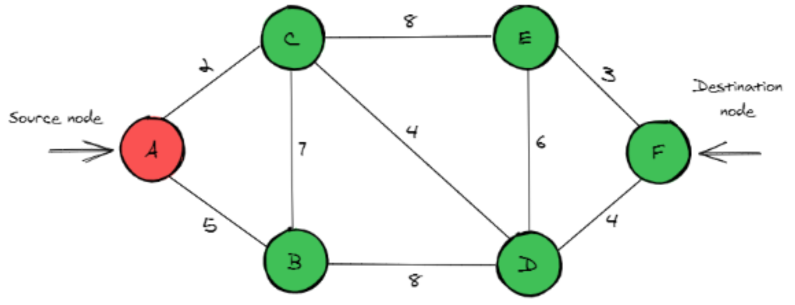
	B	C	D	E
0	∞	∞	∞	∞
0	2	2	∞	∞
0	2	2	2	6
0	2	2	3	6
0	2	2	3	6

Dijkstra Algorithm (https://algodaily.com/lessons/an-illustrated-guide-to-dijkstras-algorithm)

1.



2.



Visited
A

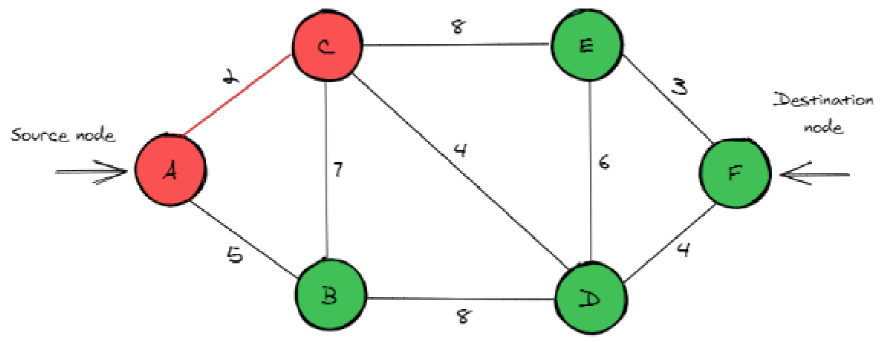
Unvisited
~~A~~
B
C
D
E
F

Shortest Distances

A	B	C	D	E	F
0	5	2	inf	inf	inf

Shortest Path: A

3.



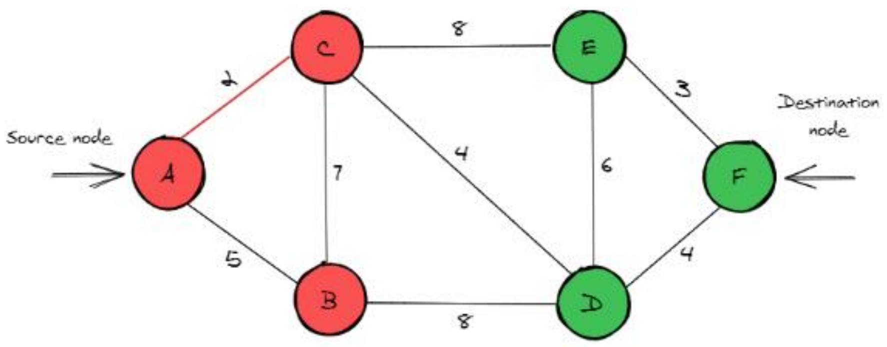
Visited
C
A

Unvisited
~~A~~
B
~~C~~
D
E
F

Shortest Distances

A	B	C	D	E	F
0	5	2	6	10	inf

4.



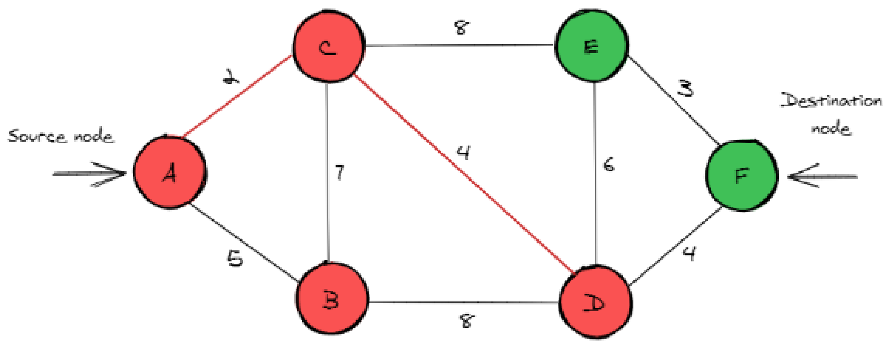
Visited
B
C
A

Unvisited
~~A~~
~~B~~
~~C~~
D
E
F

Shortest Distances

A	B	C	D	E	F
0	5	2	6	10	inf

5.



Visited

D
B
C
A

Unvisited

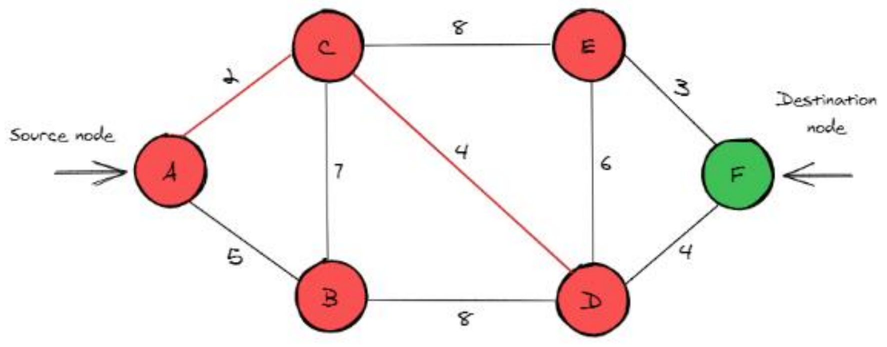
~~A~~
~~B~~
~~C~~
~~D~~
E
F

Shortest Distances

A	B	C	D	E	F
0	5	2	6	10	10

Shortest Path: A → C → D

6.



Visited

E
D
B
C
A

Unvisited

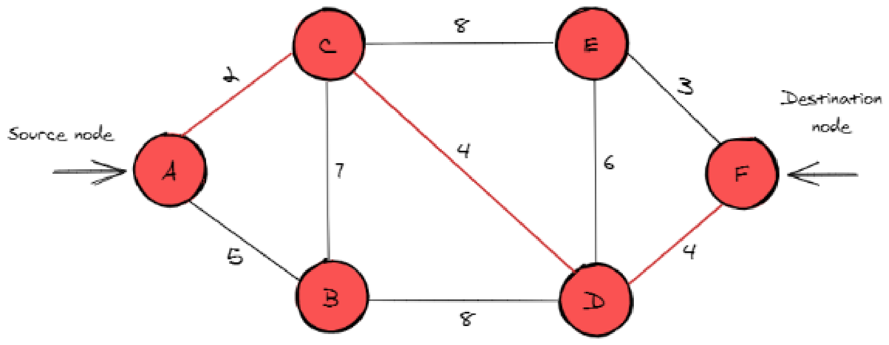
~~A~~
~~B~~
~~C~~
~~D~~
~~E~~
F

Shortest Distances

A	B	C	D	E	F
0	5	2	6	10	10

Shortest Path: A → C → D

7.



Visited

F
E
D
B
C
A

Unvisited

~~A~~
~~B~~
~~C~~
~~D~~
~~E~~
~~F~~

Shortest Distances

A	B	C	D	E	F
0	5	2	6	10	10

Shortest Path: A → C → D → F